Advanced Java Programming

Week 5 Topic Outline

Data Structures

1. What is a data structure?
    a. An implementation of an abstract data type
    b. Implementations rely on only 2 operations:
        1. Contiguous memory allocation
        2. Memory references (pointers).
2. How can we implement a list?
    a. Mutable array
    b. Linked list (singly linked, doubly linked)
3. How can we implement a stack?
    a. Singly-linked nodes pointing from top to bottom.
    b. External pointer to top node.
4. How can we implement a queue?
    a. Singly-linked nodes pointing from head to tail.
    b. External pointers to both head and tail.
    c. Insert at tail, remove from head.
5. How can we implement a map?
    a. Hash table where hash function yields index and value is stored in array.
6. How can we implement a bounded set?
    a. Boolean array (can also bit-pack)
7. How can we implement an unbounded set?
    a. Naive implementation: use a list. Requires $O(n)$ for basic `insert`, `remove`, `contains` operations.
    b. Use a hash table, storing the element itself.
8. How can we implement a sorted set?

a. Linked list + Hash table if we only care about insertion order
b. Binary search tree if we care about sorted order

9. Ponder: how can we implement a tuple? Not so easy...

Java's Collection Framework

Iterable
  Collection
    AbstractCollection
      List
        AbstractList
          ArrayList
          Stack
          AbstractSequentialList
            LinkedList
      Queue
        AbstractQueue
          PriorityQueue
        Deque
          ArrayDeque
          LinkedList
      Set
        AbstractSet
          HashSet
            LinkedHashSet
          TreeSet

Problems with Overriding Equals

1. Not using `Object` as the type parameter
2. Not overriding `hashCode` (`equals` → equivalent `hashCode` values)
3. Using mutable data in `equals` or `hashCode` computation
4. Not defining `equals` as an equivalence relation:

    a. `x.equals(null)` should return `false`
    b. `x.equals(x)` should return `true`
    c. Must be symmetric (`x.equals(y)` → `y.equals(x)`).
    d. Must be transitive (`x.equals(y)` and `y.equals(z)` → `x.equals(z)`)
    e. Must be consistent (multiple invocations yield same result)

These latter constraints cause problems when comparing subclasses and superclasses. To address these problems we define a `canEqual` method that takes an `Object` and

returns `true` if an object of this class can equal an object of the parameter's type:

```
// example
public boolean canEqual(Object other) {
    return other instanceof Foo;
}
```

Full article can be found here: http://www.artima.com/lejava/articles/equality.html